12

UCLA

OFFICE OF ACADEMIC COMPUTING

"Continued Development of Internet Protocols under the IBM OS/MVS Operating System"

FINAL TECHNICAL REPORT

TR55

Robert T. Braden

March 10, 1986

Sponsored by
Defense Advanced Research Projects Agency (DoD)
ARPA Order No. 4823
Under Contract MDA903-85-C-0130 issued by
Department of Army, Defense Supply Service-Washington,
Washington, DC 20310.

UNIVERSITY OF CALIFORNIA, LOS ANGELES

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER OAC TR55	2. GOVT ACCESSION NO. ADA167055	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) "Continued Development of Intetnet Protocols Under the IBM OS/MVS Operating System"		5. TYPE OF REPORT & PERIOD COVERED Final Technical Report 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Robert T. Braden		8. CONTRACT OR GRANT NUMBER(s) MDA 903-85-C-0130
9. PERFORMING ORGANIZATION NAME AND ADDRESS Office of Academic Computing 5628 Math Sciences Addition C0012 - UCLA Los Angeles, CA 90024		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS Order 4823
11. CONTROLLING OFFICE NAME AND ADDRESS Defense Advanced Research Projects Agency 1400 Wilson Boulevard Arlington, VA 22209		12. REPORT DATE March 10, 1986 13. NUMBER OF PAGES 61
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) Unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Distribution Unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Computer Communication Protocols. ARPANET Control Program.
Internet Protocols. TCP. IP.

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This report covers progress in the UCLA ACP, host software for the DoD Internet protocols for an IBM OS/MVS system. The Local Network Interface, the use of "C" code, other features of Release 1.6 of the ACP, and areas for future development are presented.

OFFICE OF ACADEMIC COMPUTING

University of California at Los Angeles
405 Hilgard Avenue,
Los Angeles, California 90024

"Continued Development of Internet Protocols under the IBM OS/MVS Operating System"

FINAL TECHNICAL REPORT

TR55

Robert T. Braden

March 10, 1986

Sponsored by

Defense Advanced Research Projects Agency (DoD)
ARPA Order No. 4823
Under Contract MDA903-85-C-0130 issued by
Department of Army, Defense Supply Service-Washington,
Washington, DC 20310.

"The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US Government."

SUMMARY

This document is the Final Technical Report under DARPA contract MDA903-85-C-0130, entitled "Continued Development of Internet Protocols under the IBM OS/MVS Operating System".

Under this contract, the contractor continued the development and distribution of network software for the DoD Internet protocols, to execute on an IBM 370 mainframe under the OS/MVS operating system. The central component of this network software is a subsystem called the ARPANET Control Program or ACP.

This report covers the major areas of development of the ACP under this contract: revision of the Local Network Interface, the introduction of name resolver code, and the use of the "C" language within the ACP. Finally, it summarizes the most important features of the latest ACP distribution, Release 1.6, and lists areas for future ACP development.

CONTENTS

SUMMARY ii

Chapter page

1. INTRODUCTION 1

Local Network Interface 1
Name Resolver 4
Preparation of ACP Release 1.6 5

2. LOCAL NETWORK INTERFACE 7

LNI Requirements 7
Overview of the LNI Design 16
Control and Debugging Facilities 26

3. HOST LOOKUP MECHANISMS 27

GETHBY Service 27
GETHBY Implementation 29
Multihoming and Address Sorting 31

4. C PROGRAMMING IN THE ACP 33

The Language Choice 33
Procedure Calls 35
Reentrancy 38
Storage Allocation 41
I/O System 42
Porting Difficulties 47

5. RELEASE 1.6 FEATURES 50

6. CONCLUSIONS 57

REFERENCES 60

LIST OF FIGURES

Figure page

1. Local Network Connection Using the ACP 2

2. ACP Job Components 8

3. Ptask Tree Structure 10

4. Interface Data Structures 20

5. Multiple LNI's 23

6. Host Lookup Routines 30

7. C/370 stack-frame 36

Chapter 1

INTRODUCTION

This document is the Final Technical Report under DARPA contract MDA903-85-C-0130, entitled "Continued Development of Internet Protocols under the IBM OS/MVS Operating System", in effect from 4/4/85 through 2/28/86.[1] Under this contract, the UCLA Office of Academic Computing (OAC) continued its participation in the Internet research effort of DARPA.

OAC's effort has been focused on host software to support the DoD Internet communication protocols on an IBM 370 mainframe using the OS/MVS operating system. The central component of this software is a subsystem called the ARPANET Control Program or ACP [Braden86D].

We will now sumarize the major areas of ACP development during this contract: revision of the Local Network Interface, the introduction of name resolver code, the use of the "C" language within the ACP, and the preparation of Release 1.6. Later chapters of the report will supply greater technical detail, and the Conclusion will list fruitful areas for future ACP development.

1.1 LOCAL NETWORK INTERFACE

Using the Internet Protocol (IP) [Postel81B], packets are sent from a source to a target host through one or more networks interconnected by gateways. Each IP packet is a datagram which is routed individually by the gateways, using the destination host address in the IP header. To send a segment of data, the source host prefixes it with an IP header, supplies the framing necessary for the local network to which it is connected, and sends the packet through that network to the first gateway. The gateway removes the local network framing, makes a routing decision, adds the local network framing for the next network, and forwards the IP packet. This process is repeated until the packet reaches the target host. The target host removes both the local network framing and the IP header to recover the original data segment.

[1] This contract was a follow-on to DARPA contract MDA903-83-C-0435 [Braden85A], in effect 9/1/83 through 1/21/85.

Within the UCLA ACP, the component called the Local Network Interface or LNI handles the interface to the local network. The basic LNI functions are to:

* Add the local network framing for an IP packet to be sent

* Remove the framing from a packet received from the local network

* Function as the I/O driver for the hardware interface to the local network.

Figure 1 illustrates the function of the LNI as an I/O driver for the channel interface hardware used to connect to the local network.

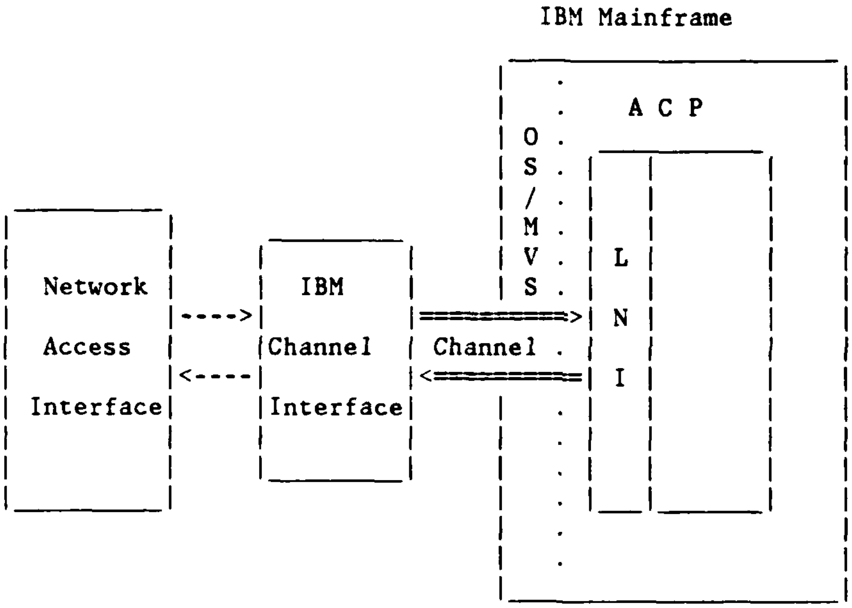

Figure 1: Local Network Connection Using the ACP

During this contract, the LNI component of the ACP was repackaged with redesigned interfaces to the rest of the ACP. The objectives of this effort were to:

* Provide Local Network Transparency

  The details of the LNI are necessarily dependent upon both the network access protocol for the local network, which defines the framing, and upon the particular I/O path used to reach the interface hardware. The I/O path might use the low-level "Execute Channel Program" (EXCP) facility of OS/MVS, or it might use an existing access method such as IBM's VTAM.

The ACP was written originally as an I/O driver for specific channel interface hardware[2] connected to an 1822 host interface on an ARPANET IMP, and using the EXCP I/O path.

The Internet transport protocol set TCP/IP provides universal connectivity, independent of the particular local network to which a host a attached. There is now a variety of local network access protocols and interfaces for ARPANET/MILNET, and some sites wish to interface IBM MVS hosts directly to LAN's, particularly Ethernets. As a result, there was a need to generalize the ACP to support different local network interfaces.

To meet these requirements, the LNI in the ACP has been repackaged to separate the processing of the local network protocol from Internet protocol processing. We refer to this separation as local network transparency, since the choice of local network interface and protocol is transparent to the code which processes IP and TCP. That is, all dependence upon (1) the particular I/O path to the network interface and (2) the local network access protocol is now isolated within the LNI code. Conversely, an LNI now makes minimal assumptions about the structure of the rest of the ACP.

This modularization of the ACP along the protocol boundary between IP and the local network will facilitate the development of new LNI's for different network connection arrangements.

* Support Multiple Concurrent LNI's

It is possible for a host computer to have two or more channel interface units like that shown in Figure 1. In that case, the ACP will need an LNI instance for each interface. There are two possibilities for these multiple paths:

1. Multihomed Host

   Each access path may have a different IP address, with the set of addresses being on the same or different networks. The host is then said to be "multihomed".

2. Multiplexed Access

   The access paths may have the same IP addresses (although the Local Network Addresses may or may not be the same). We refer to this as multiplexed access to the local network.

[2] The IF/370 channel interface built by ACC, Inc.

For example, the logical host capability [BBN1822L] of ARPANET/MILNET, which allows a set of parallel interfaces to have the same (logical) network address, could be used to provide multiplexed access. The advantages would be higher total bandwidth and increased reliability.

The repackaged LNI will support an arbitrary combination of multihoming and multiplexing of the local network interfaces, in a manner transparent to the rest of the ACP. In particular:

a) Output is load-shared among multiplexed interfaces.

b) Any interface that is down is ignored, with the load being taken by those equivalent (multiplexed) interfaces which are up.

* Document the LNI

To allow other organizations to write new LNI's, we documented the interfaces and requirements for writing new LNI modules in Technical Report TR51, "The Local Network Interface in the OS/MVS ARPANET Control Program" [Braden86A].

* Increased Efficiency

The efficiency of the LNI was improved by removing an unnecessary copy of every input packet. The IMP READ buffers are now handed directly to the Internet Protocol code.

* Improved Control and Debugging Facilities

These facilities will be listed later.

An overview of the LNI design will be presented in Chapter 2, which is adapted and condensed from TR51.

1.2 NAME RESOLVER

The Internet research community is installing a system of name resolvers and name servers [Mockap83] to replace static tables of Internet hosts. However, the static host tables are still in use in the non-research segment of the Defense Data Network (DDN).

Under this contract, we worked towards the introduction of a name resolver for host lookup functions, as an alternative to searching the static host table.

- The host lookup services of the ACP were revised to provide a compatible interface to either the existing host table lookup code or the planned name resolver module.

- Several versions of the Berkeley "BIND" name resolver code, written in "C" for UNIX,[3] were "ported" into the ACP.

  The BIND name server/name resolver package is large and complex, and porting it was not trivial. A subset of the Berkeley UNIX network I/O facilities had to be emulated, for example. This effort led to significant revisions and additions to the interface subroutines used between "C" code and the ACP.

- Fitting this "C" package gracefully into the ACP also led to a number of minor changes and additions to ACP facilities themselves.

  For example, the BIND name resolver communicates with the name server using the User Datagram Protocol (UDP). The original A-Service interface to UDP had been constructed in strict analogy with the transport-service interface for TCP, simply replacing the TCP connection with a UDP association.[4] To support the name resolver/name server code, a "datagram" UDP interface, less connection-oriented and more single-shot in character, was needed. Technical Report TR54 [Braden86B] describes the datagram-oriented interface built to satisfy this requirement.

  Chapter 3 of this report describes the new name lookup structure and the present state of development of the name resolver code. Chapter 4 describes the general problems and solutions for running "C" code in the ACP.

1.3 PREPARATION OF ACP RELEASE 1.6

The results of the development effort under this contract are reflected in Release 1.60 of the ACP, which was prepared at the termination of the contract. The features of this release are thoroughly described in the overview document:

TR48, "Introduction to the ACP -- Internet Software
for OS/MVS" [Braden86D].

TR48 is an expansion and revision of Chapter 2 of the Final Technical Report from the preceding contract [Braden85A], and replaces and obsoletes document TR36.

[3] UNIX is a Trademark of AT&T Bell Laboratories.

[4] Thus, a ULPP (see later section for a definition) was responsible for "opening" and "closing" an association.

Other new documents included with this release are:

TR49, "SMTP Implementation within the ACP" [Braden85C].

TR51, "The Local Network Interface in the OS/MVS ARPANET Control Program" [Braden86A].

TR52, "'C' Code Within the ACP" [Braden85B].

TR54, "Datagram Protocol Implementation" [Braden86B].

Release 1.6 incorporates many improvements throughout the ACP, to enhance functionality, to simplify and clarify interfaces, to improve reliability, and to fix bugs. For details, see the DOCUMENT(CHANGES) member on the distribution tape. There were two driving forces behind most of the changes: the operational experience with the ACP at UCLA and at several other sites, and the development areas listed above.

Chapter 5 describes the significant areas of change in Release 1.6.

Chapter 2

LOCAL NETWORK INTERFACE

2.1 LNI REQUIREMENTS

This section describes in a general way the functions of an LNI and its interface to the rest of the ACP, as implemented in Release 1.6 of the ACP. The next section will present an overview of the LNI structure designed to provide these functions.

2.1.1 ACP Structure

Figure 2 gives a schematic overview of ACP operation, showing the relationship of the LNI to the rest of the processing [Braden86C]. The arrows indicate the flow of data.

The major components are as follows:

* ICT

  The ACP contains a local real-time operating system called ICT (Interactive Control Task). ICT creates processes or "pseudo-tasks" (ptask for short). For each active ptask, there is a process control block called a "PTask Area" or PTA.

  The ICT services, known as P-Services, are invoked with macros described in [Stein84].

* ULPP

  For each active user/server session in the ACP, there will be at least one User-Level Protocol Process ("ULPP") ptask executing under ICT. When there are multiple ULPP's for a single session, they form a ptask sub-tree whose root is called the primary ULPP for the session.

* A-Services

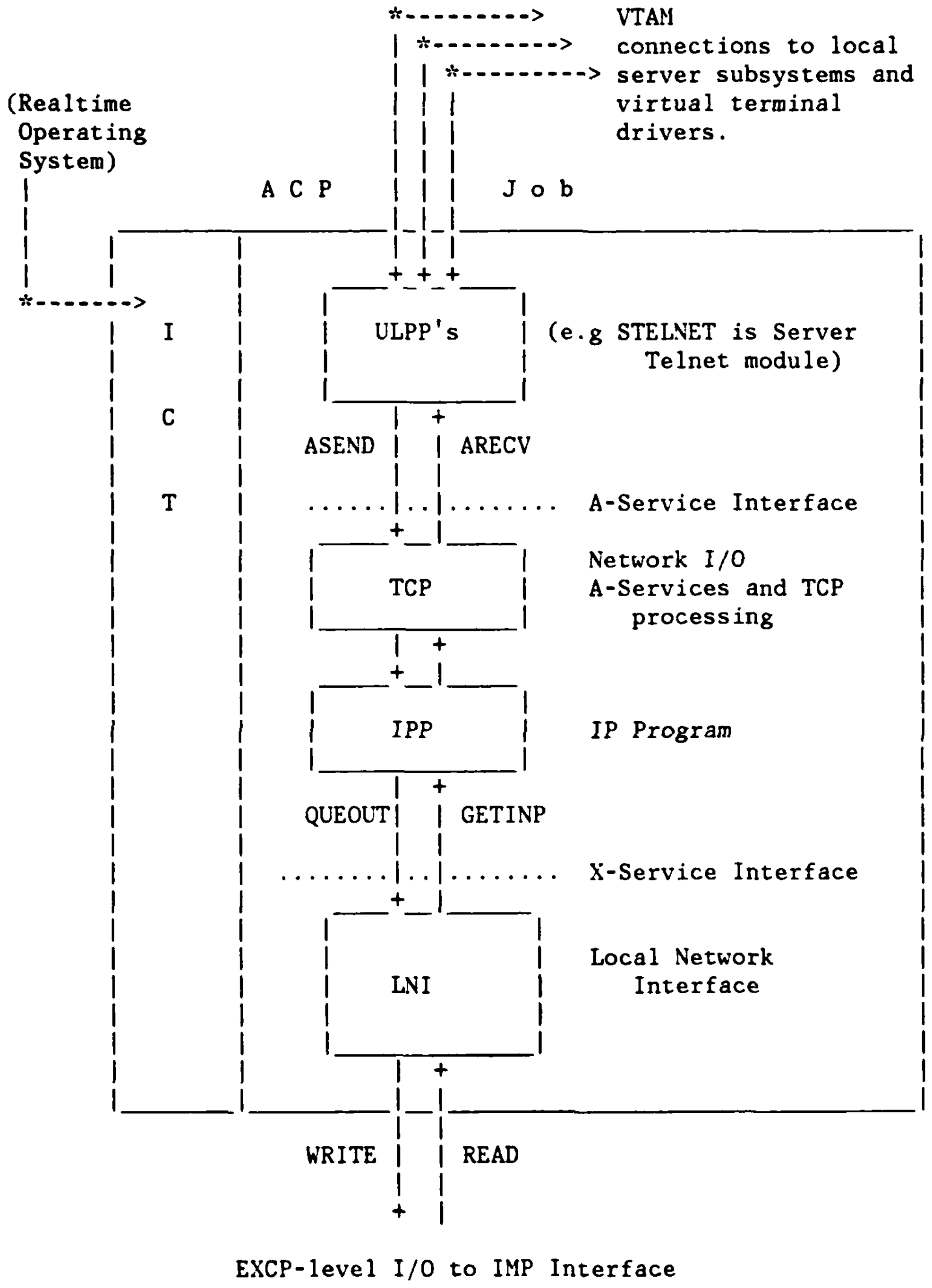

Figure 2: ACP Job Components

ULPP's obtain communication access to the Internet by issuing subroutine calls known as the A-services. A-Service subroutines are called via a transfer vector (ATRV) whose address is contained in each ULPP PTA. These calls are made using macros described in [TR21A].

For example, the ASEND service is used to send data across a (TCP) virtual circuit, while ARECV is used to receive data.

* TCP

Most ULPP's use virtual circuits created by the TCP protocol. However, there are alternative protocols at this level (e.g., UDP and ICMP).

* IPP

The IPP (Internet Protocol Program) implements the Internet Protocol IP.

* LNI

The Local Network Interface (LNI) program handles all aspects of protocol and I/O to the local network. *Figure 2 illustrated* an EXCP-level I/O interface to the network. This interface requires the caller to build chains of channel command words ("CCW's") which specify hardware READ and WRITE operations.

* X-Services

Some lower-level service subroutines and data structures are located indirectly from the ATRV via an auxiliary transfer vector called 'ACPX'. The services on ACPX, known as X-Services, are generally invoked by the ACPX macro.

The LNI services, used to invoke the LNI from the rest of the ACP, are X-Services.

Figure 3 shows the ACP ptasks which concern the LNI design.

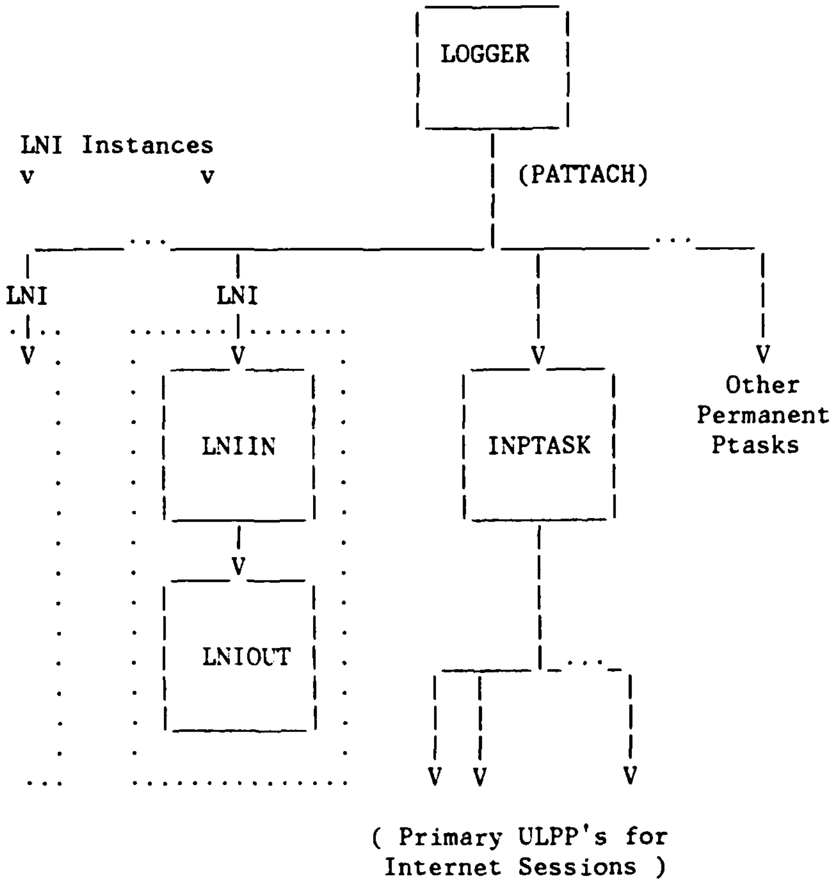

Figure 3: Ptask Tree Structure

At initialization time, ICT creates the ACP root ptask known as "Logger". Logger issues the PATTACH P-Service to create subsidiary ptasks. Repeated use of PATTACH creates a (generalized) tree structure of dependencies.

All IP and TCP processing which is not synchronous with a ULPP is performed under INPTASK; in particular, this applies to the processing of packets received from the network. IP output is processed under the ptask that originated the packet, either a ULPP or INPTASK (e.g., a TCP acknowledgment packet).

As Figure 3 illustrates, Logger creates both INPTASK and the LNI ptask(s) as direct descendants, and INPTASK in turn creates the ULPP's for active user sessions. The LNI and INPTASK ptasks are "permanent"; if they terminate Logger will restart them.

There will be an LNI instance for each channel control unit for a local network interface. Concurrent LNI instances are distinguished by an integer index (1,2,...) called the Local Network Identifier (LNID).

The local network interface hardware is normally full-duplex, allowing concurrent input and output. It is therefore convenient and natural to use a separate ptask for each direction, although a single-ptask LNI could certainly be written. We suggest that every LNI use the ptask structure illustrated in Figure 3 -- Logger creates the input ptask LNIIN, which itself creates the output ptask LNIOUT.

2.1.2 Addressing

There are two levels of network addressing of concern to an LNI.

* Local Network Address

The network access interface shown in Figure 1 is a port into a particular packet network. The network will assign to this port a unique address whose form depends upon the particular network. For example, a 24-bit address is required for an ARPANET host port, while an Ethernet interface uses a 48-bit address.

* Internet (IP) Address

The Network Access Interface also has a unique 32-bit Internet or IP address which is used in IP headers.

An IP address is divided into a network number and the "Rest" field. The latter is used to choose a particular host on the designated network. A variable-length encoding scheme is used to provide network numbers of 7, 14, or 22 bits (1, 2 or 3 bytes), leaving "Rest" fields of 24, 16, or 8 bits, respectively.

There will be a one-to-one mapping of local network addresses onto the "Rest" values of the IP addresses for that network. This may be as simple as a fixed mapping of bit fields (e.g., for the ARPANET); on the other hand, the mapping may be dynamically determined using some local protocol (e.g., Address Resolution Protocol or ARP used over an Ethernet).

The LNI is responsible for performing this address mapping. Thus, when the IP send routine hands a packet to the LNI for transmission, it passes the "Rest" field of the IP address of the target host or gateway on the local network. The LNI must map this into the actual local network address in order to build the local network framing. Upon receipt of a packet from the local network, the LNI similarly maps the source local network address (if available) into the "Rest" field of the corresponding IP address, for use by the IP input routine.[5]

2.1.3 Back Pressure

Long-haul networks such as the ARPANET generally have internal flow control and/or congestion control mechanisms which may exert "back pressure" on the host output. For example, the 1822 access protocol has a window limiting the number of outstanding packets[6] to a particular remote host. An X.25 network interface will have a similar window-based flow-control across its host (DTE) interface.

However, within the ACP the LNI provides only a reliable datagram service to the IPP. Thus, there is no (explicit) provision for flow control between the LNI and the IPP and therefore no way for the LNI to pass the network back pressure up to the IPP.

However, the design of the host-to-host protocol implementations in the ACP provides an implicit form of back pressure by imposing a limit on the number of outstanding WRE's for a given connection (or its equivalent for datagram protocols). This limit also enforces an approximate "fairness" among distinct connections to the same remote host.

2.1.4 *Software Failure Recovery*

A fundamental design goal in the ACP is robustness against software failures. The protocol processing it performs is often complex, and obscure bugs sometimes create program checks.

ICT detects a program check or other abnormal event and turns it into a pseudo-ABEND[7] ("pabend") of the particular ptask in execution. This terminates the ptask abnormally and frees all its resources, allowing ICT and the rest of the ACP ptasks to continue to execute. Ptasks can

[5] The local net address of the source of a packet is not strictly required by IP, but it is useful for providing a hint on routing return packets to the same Internet host and for trouble diagnosis.

[6] Currently the limit is 8, but a planned improvement to the IMP program will increase this limit.

[7] In IBM terminology, "ABEND" refers to the abnormal termination of an OS/MVS process or "task". The equivalent function within the ACP, terminating a pseudo-task, may be called a "pseudo-ABEND".

also be forced to terminate "normally" by system-programmer intervention (see TR42 [TR42] for details).

Referring again to Figure 3, it must be possible for any of the LNI ptasks, any ULPP, or INPTASK itself to terminate normally or abnormally without disturbing other parts of the ptask tree. The difficulty comes with pointers to buffers and other data structures which are passed between the IPP and the LNI. If the ptask that owns the storage containing these data structures terminates, the storage will be freed; the other ptask may then get a program check if it continues to process that data.

The mechanisms used in the LNI to provide cleanup in case of ptask termination are described in TR51.

2.1.5 Local Network Transparency

If a host running the ACP has more than one hardware access path to local network(s), the ACP will need multiple LNI instances.

Local network transparency is preserved in this case by a "thin layer" of program inserted between the IPP and the LNI(s). The IPP makes service requests which are independent of the number and choice of LNI's in a particular configuration, and the thin layer selects the appropriate LNI(s) for each service request. The exact function of the thin layer will be described below.

2.1.5.1 Subsetting Local Network Functionality

In general, the IPP uses only a subset of the services provided by any particular local network. For example, all IP traffic uses a particular value (155) of an ARPANET parameter called the "Link Number". The question is: where should the LNI parameters be bound in order to subset the network services, in the LNI or in the IPP?

In order to provide transparency the IPP/LNI interface must be the same for all networks. Therefore, each LNI must map that interface into the appropriate subset of its capabilities; thus, an ARPANET LNI must set the link number to 155. This implies that the free parameters must be bound within the LNI.

On the other hand, when we write an LNI for a local network we want to provide a completely general interface to that network's capabilities, to allow for future developments and experiments.

As a solution to this dilemma, the LNI has been designed with both (1) a standard (transparent) interface for IP and (2) an extended "raw network" interface for future applications. Thus, the parameter area used for sending an output packet to the network includes an "extended interface" flag bit. This bit is off for the standard interface. If the bit is on, the parameter area is extended to include the additional "raw" parameters (e.g., the link number for an 1822 interface to ARPANET). The standard parameter list is universal for all LNI's, but the format of the extended parameter list is necessarily dependent upon the particular local network access protocol.

2.1.5.2 ARPANET Uncontrolled Packets

The IMP subnetwork is capable of sending messages with subnormal reliability and subnormal delay using "Subtype 3" or "Uncontrolled" mode (see Appendix A). The Uncontrolled mode has not been extensively used because it poses the threat of uncontrolled congestion in the network, but future IMP software development may make the use of Uncontrolled packets desirable.[*]

In the Internet Protocol model the choice of message subtype (and any other network parameters [Poste81B]) is be based on a field in the IP header called Type of Service (TOS). It is supposed to be used by the hosts and gateways to select appropriate parameters for each network which is the packet traverses.

The LNI provides for Subtype 3 packets while preserving local network transparency by including the TOS in the standard parameter area used for network output. An appropriate subset of values of the TOS field will cause the 1822 LNI to send a Subtype 3 packet.

However, Subtype 3 may only be used if the packet is shorter than 113 bytes. Thus, the maximum packet size ("Maximum Transmission Unit" or MTU) is dependent upon the TOS chosen. To make general use of Subtype 3, the TCP and IP layers would need a more dynamic way to choose the MTU than they have presently; there is currently a single MTU per LNI. Fortunately, planned improvements in the IMP software will remove the low MTU for Subtype 3 packets.

For use in a military environment, the precedence bits in the TOS also need to be mapped into appropriate parameters for the local network.

[*] The original ACP implementation of TCP/IP was designed to support uncontrolled messages primarily. There were mechanisms for selecting the message type on a per-host basis, for example. To date, all ACP operation has used standard messages.

2.1.5.3 ISO IP

A future version of the ACP may support the ISO IP protocol [ISOIP] in addition to the present Internet IP. ISO IP will use a different ARPANET link number, but otherwise should use the standard (transparent) interface parameters. However, the standard interface will have to be reformatted for ISO IP to accomodate a variable-length destination host address.

We suggest that a new flag bit be defined in the standard interface to indicate ISO IP.

2.1.5.4 NMC Intercept/Packet Trace

The LNI within the ACP has historically had a network-level packet intercept mechanism known as the Network Measurement Center (NMC) Intercept. The A-Service ANMOC may be used to define a filter to select which packets are to be copied into a circular buffer. A particular filter element may be active for input or output or both. The filter selects only on the local network header.

The NMC intercept is used by the packet trace display program ACPEEP [TR42]. ACPEEP may be used to trace packets at the local network level, the IP level, or the TCP level. For the higher-level traces, ACPEEP calls ANMOC to set up an NMC Intercept filter that captures all IP data packets; ACPEEP then filters the packets itself.

Note that ACPEEP is passed the entire packet as received from the local network, including the local network framing (leader) and possibly the IP header, TCP header, and user data. ACPEEP must know how to parse all these header levels, and ACPEEP therefore must contain local-network-dependent code. To enable ACPEEP to parse alternative formats of local network framing, the NMC Intercept mechanism sets a 'leader format' byte in the header which precedes each captured packet.

The facilities of ACPEEP have been extremely useful for diagnosis of problems at all protocol levels, and we would strongly recommend that the developer of a new LNI include the NMC Intercept mechanism and extend ACPEEP to parse the new network framing.

Although ACPEEP is to remain explicitly network-dependent, the ANMOC service has been modified to achieve local network transparency. The local-network-dependent part of the processing was moved into the LNI (see the SETNME service below).

There is a problem in designing the ANMOC parameter list NMPARM: we want to be able to specify filters for IP in a form independent of the particular local network format; conversely, we want to be able to specify arbitrary filters for particular local networks. This problem is analogous to that discussed earlier under "Subsetting Local Network Functionality", and an analogous solution was adopted. NMPARM may specify either "standard" filter parameters, appropriate to IP and universal to all LNI's, or else "raw" filter parameters, which are necessarily dependent upon the particular LNI and network header.

2.2 OVERVIEW OF THE LNI DESIGN

2.2.1 X-Service Calls

We begin by describing the X-Service calls to the LNI. These calls define the input and output interfaces to the LNI(s) from the rest of the ACP (in particular, from the IPP).

All of these calls are pseudo-disabled, i.e., no other ptask is allowed to execute while they are in progress.

2.2.1.1 QUEOUT -- Queue Output Request

```
    [ label ]  ACPX  QUEOUT,wre-address
```

To send a packet, an ACP routine (normally, the IPP) builds a parameter list and queue element called a Write Request Element or WRE, and calls ACPX QUEOUT.

The WRE defines the data to be sent, the destination host address (as the "Rest" field of the IP address), and the source host address. The host addresses are specified indirectly by the address of a control block called an "ICB" [Braden86C]. The WRE defines the data to be sent by a list of (address, length) pairs or extents. Thus, output is transmitted to the LNI by reference, and the LNI must perform a gather-write operation.

The QUEOUT call will add the WRE to an internal LNI queue and if necessary awaken the LNIOUT ptask to send the data. The "Complete" flag bit (WREF2COM) in the WRE will be turned off by QUEOUT, and it will be turned back on by LNIOUT when the data has been sent.

The WRE may also specify the address of a Write Completion Exit routine, which the LNI will call when the data has been sent. This exit routine can include code to signal a semaphore in the ptask that called QUEOUT.

The caller must not change or reuse the WRE or any of the packet buffers until the WRE has been marked "Complete", with one exception: the caller may turn on the "Purged" bit (WREF1PRG) at any time. If the WRE has not yet been sent when the Purge bit is turned on, it will not be transmitted when its turn comes, but will be marked "Complete" (and Purged) and the Write Completion exit will be called normally.[9]

If the caller needs to free the storage containing the WRE or packet buffers, it must first call the HALTIO service routine see below), specifying the ICB address contained in the original WRE.

2.2.1.2 HALTIO -- Purge Output Requests

```
[ label ]  ACPX  HALTIO,argument
```

This service is required to allow a ptask to terminate safely while it has pending WRE's queued in the LNI, or has a busy input buffer. HALTIO has different calls for purging input and output operations.

* Argument = ICB Address

All WRE's specifying this ICB address will be dequeued and marked "Purged".

* Argument = complement of P3CB Address

The READ buffer which is currently "busy", i.e., has been handed to INPTASK via a GETINP call, will be freed.

[9] This facility was inserted to provide TCP with a way to suppress redundant empty-ACK packets while they are waiting in the LNI output queue.

2.2.1.3 GETINP -- Request Input Buffer

```
[ label ]  ACPX  GETINP,Prev-buff-addr,P3CB-addr
```

This service returns in R1 the next input buffer address, or zero if there is no new buffer available.

Each call to GETINP has the side effect of freeing the buffer returned in the previous call; this buffer address is 'Prev-buff-addr' which must be specified as zero in the initial call to GETINP.

Whenever a new input packet becomes available, the LNI will awaken the IPP ptask (INPTASK) by issuing a PPOST for its INPUT semaphore. When the IPP ptask is awakened, it will call GETINP repeatedly and process each buffer which is returned. When GETINP returns zero, all available input buffers will have been processed and freed.

Note that the buffers from the LNI headware READ pool are passed directly to the IPP, and that the IPP must process these buffers one at a time, promptly, and in order. This allows the LNI to use a simple ring of hardware READ buffers and to pass them directly to the IPP without copying a buffer. In fact, the IPP immediately copies the packet into a reassembly buffer, which may be queued at the IP or TCP level; the LNI READ buffer is then immediately freed (by the next call to GETINP).

The buffer addresses in this call actually point to a buffer header, which is followed in storage by the complete packet as received from the network interface hardware. Within the buffer header there is a flags field. If the "Host Dead" flag bit is on, there is no data in the buffer; instead, it is an asynchronous indication from the network that a previously-sent packet was undeliverable. The IPP will note the bit and immediately call GETINP for the next buffer.

2.2.2 SETNME -- Setup NMC Intercept Filter

```
[ label ]  ACPX  SETNME,NMPARM-addr,NME-addr
```

This service is used internally by the ANMOC A-Service to perform the local-network-dependent portion of setting up a packet intercept filter.

2.2.3 LNI Module and Data Structures

This section describes the general structure of an LNI module and the control blocks used to interface an LNI with the rest of the ACP.

The LNI code and data are packaged in a separate load module. The name of this module is specified in the installation configuration module ACPCONFG [TR42]; we will use the generic name LNIMOD in this chapter.

Figure 4 illustrates the principal communication areas used between LNIMOD and the rest of the ACP.[10]

* LNIROOT

  LNIROOT is a non-volatile data area which is assembled into the ACP resident module ARPAMOD. LNIROOT consists of a set of double-word "slots", numbered 0, 1, 2,... Each LNI instance is assigned a unique LNID, which also designates its slot number in LNIROOT.

  There is no LNID=0; slot 0 of LNIROOT contains the address of the ptask (INPTASK) whose INPUT semaphore the LNI must PPOST when a packet is received. Beginning with slot 1, each slot contains: the address of the LNI Control Area for the LNI instance; flags which indicate the up/down status of the LNI; and a 3-byte area used by the LNI as non-volatile storage.

* LNICA

  For each LNI instance, there is a control block known as the LNI Control Area or LNICA, which is link edited into LNIMOD.[11] The LNICA address must be the entry point of LNIMOD.

[10] This Figure omits one small complexity: the Logger Control Area pointer is actually indirect through the ACPX transfer vector.

[11] LNIMOD cannot be reentrant if the LNICA is link edited with it. Lack of reentrance is not a serious a problem; to configure an ACP with with two instances of the same network interface type, simply link edit two copies of LNIMOD with different names.

However, this is only a choice of convenience; it is certainly possible to write a reentrant LNI which creates its LNICA dynamically.

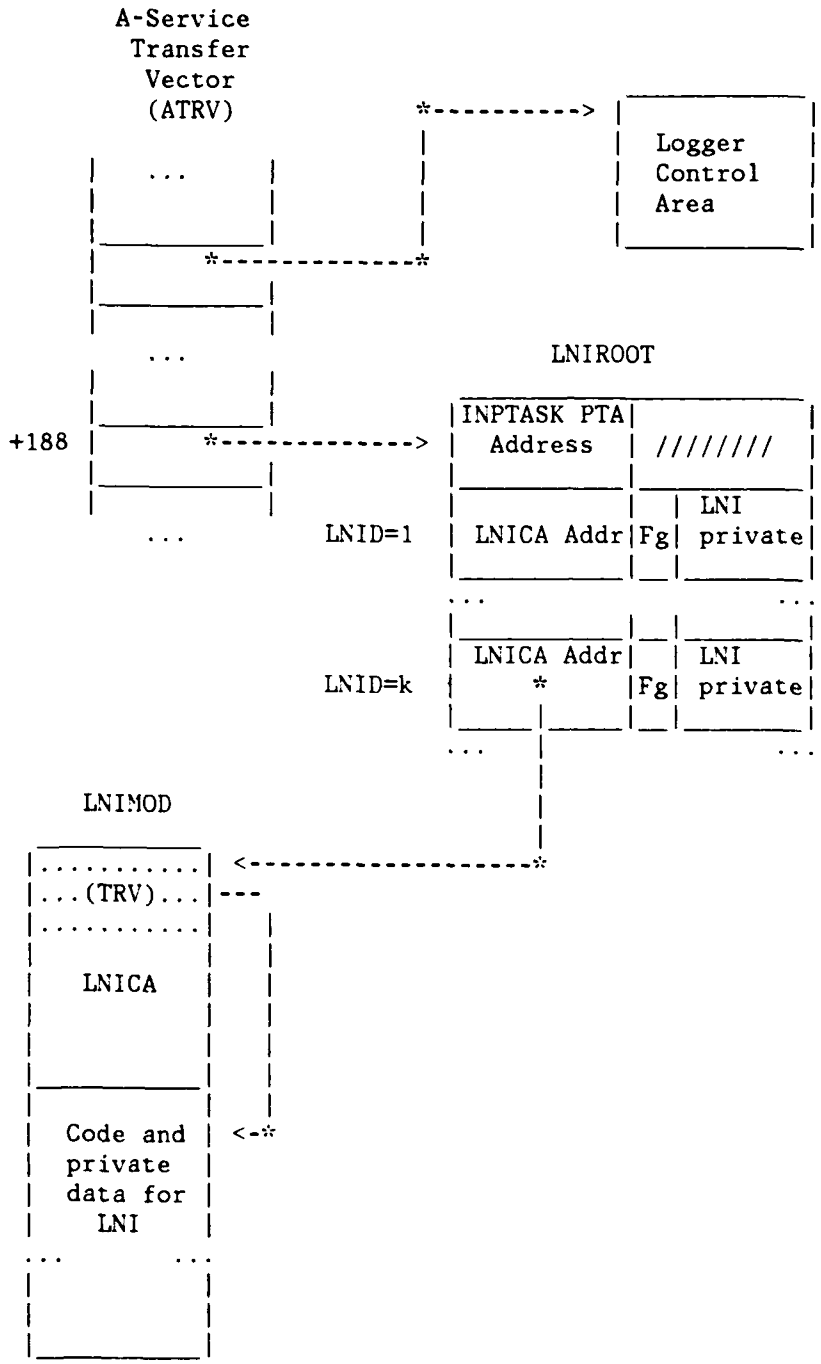

Figure 4: Interface Data Structures

The LNICA begins with a transfer vector, used to reach the LNI service subroutines that other ACP modules may call. No ACP routine outside LNIMOD depends upon the LNICA format, except for the transfer vector.

When LNIMOD is PATTACHed by Logger, execution of the subptask begins at the LNIMOD entry point, which is the transfer vector word at offset +0. This word must contain a branch to the LNI initialization routine.

* Logger Control Area

The Logger Control Area is a resident communication area. In particular, it contains the Logger Flags (LFLAG) and the Logger PTA address (LOGTPA).

Logger Flag bits are used to pass execution-time parameters to the LNI(s), to signal the rest of the ACP when the up/down status of the interface changes, and to request control and status functions from the LNI(s).

One of the functions of the LNIMOD initialization program is to plant the LNICA address into its slot in LNIROOT. Furthermore, it must establish a PDEBUG exit to clear out this pointer if the LNIIN ptask terminates.

To invoke an LNI service subroutine, an ACP routine codes an an ACPX macro to call an X-Service routine. The X-Services are implemented as <u>resident stubs</u>. These resident stub routines use the pointer chains illustrated in Figure 4 to locate and call the appropiate LNI service subroutine(s) through the LNICA transfer vector.

However, the resident stubs are more than simple indirect linkages.

* They handle the case of an LNI (instance) not active, or the corresponding interface not operational.

When a particular LNID is selected, the resident stub checks that the corresponding slot in LNIROOT has a non-zero pointer to an LNICA; if not, the LNI is not active, and the stub must respond appropriately. Even if the LNI is active, its flag byte in LNIROOT may indicate the corresponding interface hardware is malfunctioning; in this case, too, the stub will abort the action.

* The stubs hide a mechanism which makes the rest of the ACP independent of the LNI configuration.

In particular, they hide the use of multiple LNI's from the X-Service callers, using a mechanism which will be described in the next subsection.

Thus, the resident stubs constitute the "thin layer" between the IPP and the LNI which was mentioned earlier. In some cases the stubs alter the calls, so there are slight differences between the LNI service calls to the resident stubs and the calls as finally seen within LNIMOD.

2.2.4 Multiple LNI's

Within the ACP, there is a control block known as the P3CB for each IP address by which the ACP is known [Braden86C]. The P3CB contains the local IP address (field P3MIHOST) and also the set of LNID's of corresponding LNICA's. This set is restricted to a range of adjacent integers, and the P3CB specifies the set by giving the first and last in the range. For example, in the simplest case of a single LNI, the P3CB will specify the LNID range (1,1).

The address of the appropriate P3CB is a parameter to the calls to the LNI service stubs. These stubs use the LNID range to choose the the appropriate LNID(s). This is illustrated by Figure 5. The configuration information shown in the P3CB's in this Figure is taken directly from ACPCONFG at ACP startup time.

```
 (1)  MULTIHOMED  HOST

      LNICA's
                                     P3CB's
  ______________
 |              |                 ______________
 |              |                |              |
 |   LNID=1     |                | MIHOST=      |
 |              |<==========>    |  10.1.0.1    |
 |              |                | LNID range=  |
 |              |                |  (1,1)       |
 |______________|                |______________|


  ______________
 |              |                 ______________
 |              |                |              |
 |   LNID=2     |                | MIHOST=      |
 |              |<==========>    |  26.5.5.5    |
 |              |                | LNID range=  |
 |              |                |  (2,2)       |
 |______________|                |______________|



 (2)  MULTIPLEXED  ACCESS

      LNICA's
  ______________
 |              |
 |              |
 |   LNID=1     |
 |              |<====**        P3CB
 |              |     ||
 |              |     ||         ______________
 |______________|     ||        |              |
                      **====>   | MIHOST=      |
                                |  10.1.0.1    |
  ______________                | LNID range=  |
 |              |     **====>   |  (1,2)       |
 |              |     ||        |______________|
 |   LNID=2     |     ||
 |              |<====**
 |              |
 |              |
 |______________|
```

Figure 5: Multiple LNI's

2.2.5 Resident Stub Algorithms

We can now describe the algorithms used by the resident stubs to hide the existence of multiple LNI's from the rest of the ACP, and define more accurately the services provided by the LNI service routines QUEOUT, GETINP, HALTIO, and SETNME.

2.2.6 QUEOUT Service

LNI Service: QUEOUT(WRE)

When ACPX QUEOUT calls the resident QUEOUT stub, the WRE will contain the ICB address. The resident stub in turn calls a QUEOUT service routine in an LNI, passing the same WRE. The LNI will treat the ICB address in the WRE as an uninterpreted "purge handle" for use in HALTIO.

The QUEOUT stub locates the P3CB from the ICB and saves (in the P3CB) the last LNID used for output. Using this value and the LNID range, it rotates among the LNID's in the range for successive calls. This provides equitable load-sharing across equivalent interfaces in a manner transparent to the rest of the ACP.

If a selected LNI instance is not active or not up, the stub tries the next LNID. If none of the LNI's in the range are active and up, the stub returns an error code (8) in R15.

If the WRE address is zero, the stub assumes it is a control/information call, and passes it to ALL of the LNID's in the range. The LNI's then examine bits in the Logger Control Area to determine the function that is desired.

2.2.7 GETINP Service

LNI Service: next-buffer = GETINP

The GETINP stub calls each of the GETINP services for the LNID's in the range of the given P3CB, until it finds an available input buffer. It returns this buffer, or zero when no LNI in the range (is active and up and) has a buffer available.

2.2.8 HALTIO Service

```
LNI Service:  HALTIO( purge-handle )
                  or
              HALTIO( neg-number )
```

When it is called with: ACPX HALTIO,<ICB address>, the resident stub uses the ICB address to locate the P3CB; when it is called as ACPX HALTIO,<-P3CB address>, the stub complements the argument to obtain the P3CB address.

The HALTIO stub loops through ALL of the (active and up) LNI's in the range of the P3CB, calling the corresponding LNI HALTIO service routines with the original argument as a parameter.

Given a negative parameter, the LNI HALTIO service frees a busy input buffer. If the parameter is not negative, the LNI HALTIO service treats it as an uninterpreted "purge handle", removing from its output queues all WRE's containing that value in their "purge handle" (ICB address) field.

2.2.9 SETNME Service

```
LNI Service:  SETNME( NMPARM-list, NME-addr )
```

This service is used to set up a filter element (NME) for the NMC intercept. SETNME contains the local-network-dependent parts of the ANMOC service.

The resident stub obtains the LNID number from the NMPARM list and simply calls SETNME in that LNI.

2.3 CONTROL AND DEBUGGING FACILITIES

Finally, we will mention the various facilities for controlling and debugging an LNI which are built into the existing IMP 1822 LNI. We strongly recommend they be carried into any other LNI implementation.

* Software Loopback

The LNI (and therefore, the ACP) can be operated without any channel interface hardware, looping output packets back to the input within the LNI. This provides a very powerful debugging facility for the ACP, which can be executed under the TSO TEST processor in software loopback mode.

The loopback is implemented at the lowest possible level -- the WRITE channel program is interpreted and used to build a packet in a READ buffer. Therefore, it provides a reliable test of the entire ACP mechanism (except for timing effects, of course).

Specifying IMP=NO in the execution parameters used to start up the ACP will put the LNI(s) into software loopback mode.

* Test Mode

To test the channel interface hardware, it is useful to physically loop back the IMP side of the interface. This is possible because the 1822 protocol is symmetrical between input and output, except for the flow-control messages (RFNM's) which normally flow from the IMP to the host.

To support this testing requirement, the LNI has "test" mode, in which RFNM's are not required. Specifying TEST=YES in the execution parameters used to start up the ACP will put the LNI(s) into test mode.

* Tracing I/O

For debugging the LNI, it may be useful to keep a full trace of all local network packets sent and received. The trace is sent to the TEST log dataset, and it is enabled by specifying TRACE=YES in the execution parameters used to start up the ACP.

* Packet Trace

We have already described the packet trace program ACPEEP.

Like the software loopback, the packet trace operates at the lowest (channel command chain) level. Unlike software loopback, the packet trace mechanism incorporates a filter to selectively trace packets and operates on both the input and output streams.

Chapter 3

HOST LOOKUP MECHANISMS

This chapter will describe the changes which were made to support the new domain names and name server/name resolver system [Mockap83]. The general approach was to build a new name lookup service in a form appropriate for a name resolver and then recast the existing mechanism as a subset.

3.1 GETHBY SERVICE

A new host lookup A-Service, GETHBY, ("Get Host by..."), has been implemented [TR21A]. The basic GETHBY forms:

GETHBY NAME: Maps Host name -> Host number

GETHBY NUMBER: Maps Host number -> Host name

are modelled on the lookup routines gethostbyname() and gethostbyaddr() which are included in the BIND name server package for Berkeley UNIX.

GETHBY provides a more general interface for host lookup than the previous IHOST service. GETHBY returns the principal host name, alias names (nicknames), and multiple IP addresses (for a multihomed host) in one call. This generally corresponds to the information which may be returned in the response to one host name query to a name server. GETHBY is now considered the fundamental service, so the IHOST NAME and IHOST NUMBER services have been rewritten to call GETHBY internally.

GETHBY returns the address of a 256-byte area obtained with PCORE and called a HOSTENT(ry) block. A HOSTENT block contains a vector of pointers to name strings, the name strings themselves, and a vector of IP addresses. The caller is generally expected to issue PCORE FREE for the area after using the information.

The forms of GETHBY are as follows:

* GETHBY NAME,hostname-addr

  The argument is the address of a host name string, which is terminated by a zero byte. The name can be the principal name or a nickname for the host, but an exact match is required. Case is irrelevant.

- * GETHBY ADDRESS,host#-addr,host#-length

  The arguments are the address and length for a 32-bit IP host number. The length should always be 4; it is included to allow future generalization to other classes of host addresses.

- * GETHBY NEXT,last-HOSTENT-addr

  This form may be used to sequence through some set of host entries. In the host table version, this set is all hosts in the table. In the name resolver version, this command could be used to display the current cache contents (but has not yet been implemented).

  The argument to each call is the HOSTENT pointer returned by the preceding call, or zero in the first call.

- * GETHBY UNAME,hostname-addr

  Similar to GETHBY NAME, UNAME will return the unique name which is the "best" match, if no exact match can be found. In the name resolver case, this will result in a CQUERYU query.

- * GETHBY MNAME,hostname-addr

  MNAME is similar to UNAME, but MNAME may return multiple names which begin with the specified argument string. In the name resolver case, this will result in a CQUERYM query.

  If this call returns multiple names, each will be in a separate HOSTENT block, and the HOSTENT blocks will be chained into a list using an internal word.

Each of these calls has a single argument, which is passed to the GETHBY A-Service in R1. R0 contains a value which implies the call form:

R0= n > 0:	GETHBY NUMBER, n= length (always =4)
R0= -1:	GETHBY NAME
R0= -2:	GETHBY NEXT
R0= -3:	GETHBY UNAME
R0= -4:	GETHBY MNAME

3.2 GETHBY IMPLEMENTATION

The GETHBY A-Service routine is actually a resident stub that calls a transient lookup module; see Figure 6. The A-Service uses PCORE to obtain a stack area and PLOAD to fetch the transient module, and then calls the lookup module at its entry point (also named GETHBY).

The calling conventions for the transient module are compatible with "C" procedures (see the next chapter and especially Figure 7). Thus, the parameters are passed in the beginning of the first stack frame.

The USMTP module, which is written in "C", calls GETHBY using a C-compatible call. This means that the transient host lookup module could be link-edited with USMTP. This might seem appropriate because USMTP will be the most frequent consumer of GETHBY services. However, it has two possible disadvantages: both modules are very large, and they would use the same stack. We have therefore chosen at present to link edit with USMTP a trivial BAL stub routine, named GETHBY# but with entry point GETHBY, that turns the C-compatible call into an A-Service call. This is illustrated in Figure 6.

There are two alternative but compatible transient lookup modules; configuring an ACP installation with one or other other will require little more than a module rename.

* GETHBY -- Host Table Version

Release 1.60 is distributed with a transient lookup version (also named GETHBY) that searches the existing static name tables. These tables are linkedited with the transient GETHBY routine, as Figure 6 illustrates.

In previous ACP versions, the AHSCAN A-Service provided a database access method for searching the static host tables. Since it is dependent upon the host tables, AHSCAN has been withdrawn as an A-Service (although the code is actually incorporated into the GETHBY transient lookup module).

The only ACP routine which used the AHSCAN service was OISUMH (NETSTAT host summary display) within module OPFORMAT; this has been rewritten to use GETHBY instead. The new OISUMH has additional functionality, displaying all the names and numbers associated with a specified host. The command to display all hosts uses GETHBY NEXT.

* GETHOST -- Name Resc'ver

The alternative transient lookup module, named GETHOST, is a name resolver. GETHOST is implemented as a "C" module, adapted from the Berkeley BIND routines listed above. It has not been distributed in Release 1.60.

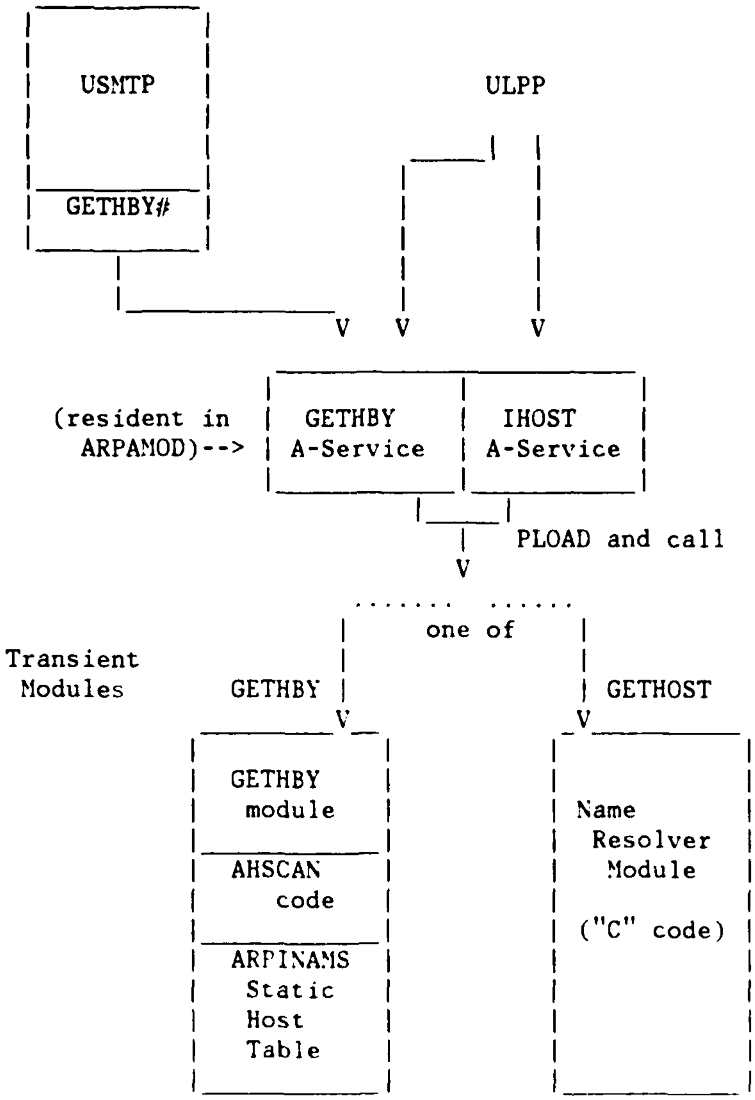

Figure 6: Host Lookup Routines

The BIND package includes both a name server and a name resolver, which are assumed to execute on the same host. The name resolver is very simple, having no cache or recursive lookup capability. Instead, the resolver sends a query to the local name server, which does have these capabilities and acts as an agent for the resolver.

The name server executes under a permanent ptask named NAMSERV. GETHOST formats name server query packets and sends them to NAMSERV using UDP, and NAMSERV returns the result via UDP.[12]

The partial-match forms UNAME and MNAME have not yet been implemented in either the table or the name resolver version. Both require the specification of a target domain, which corresponds to the tail (right-hand end) of the domain name string. We propose that this specification be made either explicitly or implicitly.

- * The parameter string to GETHBY may be of the form:

  <arg name>?<target domain>

  to specify the target domain explicitly. For example,

  GETHBY UNAME,"oac?edu"

  should return a HOSTENT for "oac.ucla.edu".

- * If no argument string appears, a default argument will be taken from ACPCONFG. This will allow efficient lookup of hosts within the local domain. For example, at UCLA the default target would be "ucla.edu".

In the static table version, the AHSCAN routine will have to be able to search for a partial match with both the head and the tail of the domain name.

3.3 MULTIHOMING AND ADDRESS SORTING

Internet electronic mail is generally delivered using addresses of the form:

user @ host

where 'user' is some string defining a mailbox at the Internet host with (domain) name 'host'. The User SMTP program (USMTP) must map each such host domain name into a 32-bit Internet host number in order to deliver the mail.

Some hosts are multihomed, which means that the name-to-address mapping will yield more than one 32-bit address. Multihoming is something of an embarassment to the Internet architecture, since it assigns multiple addresses to the same entity (host). However, for a variety of reasons

[12] As a later optimization, it would be desirable to short-circuit the network I/O for these UDP packets when source and destination hosts are both local.

multihoming is not unusual in the Internet.

How is USMTP to handle a list of 32-bit addresses? One of the reasons for multihoming is to provide reliable service when one network interface is down. For this to succeed, USMTP must be willing to try the alternative addresses, one after another, until one of them succeeds in accepting the mail.

However, in many cases there is a preferred order for trying the addresses. In almost every case, it is better to use an address on a network to which the sender is also connected, if possible. If there is a choice of long-haul and LAN networks, it will generally be preferable to try the high-bandwidth LAN first.

In general, the best ordering depends upon the network numbers of the addresses and must be chosen with a knowledge of administrative and technical constraints in the Internet neighborhood of the sender. If the target is many networks away from the source, it is impractical and probably unnecessary to worry about the choice of order.

These considerations determined the provisions in the ACP to handle multihomed hosts:

* USMTP calls GETHBY NAME to map a target host name into its address(es). GETHBY returns up to four alternative IP addresses for the host.

* The ACP configuration module ACPCONFG contains a fixed table of network preferences. GETHBY uses this table to sort multiple addresses into a preferred order. Normally, the network local to the ACP should appear first in this table.

* USMTP then tries each of the the alternate IP addresses of a multi-homed host, in turn, until it is successful in sending the mail.

Chapter 4

C PROGRAMMING IN THE ACP

4.1 THE LANGUAGE CHOICE

The first version of the UCLA ACP was written (starting in 1970) in IBM Basic Assembly Language (BAL). Since that time the ACP has evolved continuously, adapting to both a different operating system environment (OS/MVS replacing OS/MVT) and a new host-to-host protocol (TCP/IP replacing NCP). However, the basic ACP structure has been maintained throughout this period, and until recently BAL has been the only programming language used within the ACP job.

Although the advantages of a higher-level language over assembly language are well known and undeniable, there were a number of reasons for our remaining with BAL for so many years.

* OAC had available a highly-skilled stafff of system programmers, comfortable with writing and maintaining complex system programs in BAL. Furthermore, program debugging and diagnostic tools were included in the ACP to make BAL programming relatively effective.

* Prior to the installation of a virtual memory operating system (OS/MVS) at UCLA in 1978 there was a severe main storage constraint on the ACP. Programs written in a higher-level language typically require three to five times as much main storage as the equivalent BAL program, which made the use higher-level languages impractical.

* There were no good candidates for languages or compilers.

  IBM refused to release to customers their PL/I-like development language. Wirth's PL/360 could have been adapted, but it would have required the development of new system interfaces and had other serious limitations.

  PL/I was used extensively for developing components of the National Software Works (NSW) code that ran under TSO and outside the ACP region. However, PL/I has some severe disadvantages for use within the ACP. The most important problem is the large and complex runtime environment used by PL/I; to modify that environment would have exposed us to a maintenance problem with every compiler release.

Under OS/MVS, the ACP executes in its own virtual memory address space, currently about 2M bytes. The relaxation of the memory constraint and the requirement to develop new user-level protocol modules (e.g., SMTP sender and receiver) led to a reconsideration of the introduction of higher-level language programming into the ACP.

Over the past year, we have incorporated into the ACP new modules written in the "C" programming language [KernRi78] and compiled using the AT&T C/370 compiler. This effort has been highly successful, and we would hope that many future extensions to the ACP will be done in "C" rather than BAL [Braden85B].

The "C" language and the C/370 compiler have several attributes favoring their use within the ACP.

* The "C" language is heavily used within the DARPA Internet research community, giving it an important advantage over any other choice (e.g., Pascal, PL/I, PL/360, ADA, ...): there is a lot of relevant software available in "C". Furthermore, the designers of "C" paid careful attention to the requirements for writing portable code [JohnRi78].

  As a result, we have been able to "port" several large and important "C" programs from a VAX environment into the ACP with little effort and few changes.

* "C" is really a higher-level assembly language; i.e., it falls between assembly language and a true higher-level language like Pascal or PL/I. With "C" the prgrammer is still close to assembly language, since "C" does not interpose a complex environment. The time and space costs of using C/370 are significant but tolerable for many applications.

* Most of the "C" environment consists of a library of subroutines which are themselves written in "C" [KernRi78]. The general structure of this library is fairly well documented and the routines are easy to modify and adapt as necessary.

* The structure of the C/370 compiler lent itself to modification of the environment. In particular, C/370 produces assembly language which is assembled by the standard IBM Level H assembler in a later jobstep. This allowed us to modify the procedure call environment for a "C" module by simply changing BAL macros used in the assembly step.

This chapter will describe some of the design choices and programming rules used to incorporate "C" code into the ACP.

4.2 PROCEDURE CALLS

The C/370 compiler generates code for a linear execution-time stack. Since the natural coding style in "C" makes heavy use of subroutine calls and even recursion, a linear stack is the only reasonable choice.

C/370 requires that the stack be a single contiguous segment, and of course it must be large enough for the deepest call chain. Within the ACP, we normally allocate a 15*256 byte (3.8K) stack area for each "C" execution instance. This is more than adequate for most programs. However, a program which requires a larger stack[13] can call the subroutine _bigstak() to expand the stack.

Figure 7 shows the format of a single "stack frame", the region of the stack used by a particular procedure invocation. In this picture, the stack grows downward.

Fortunately, the C/370 compiler produces code with procedure call conventions which are nearly compatible with normal OS/MVS call conventions, and hence with the ACP. In particular, upon entry to the subroutine:

R13 points to an area into which registers may be saved.
R14 is a return address
R15 is the address of the called subroutine.

However, there are two compatibility problems: (1) parameters are passed differently, and (2) within the ACP, R11 must contain the PTA address before an ACP service can be called.

The solutions to these problems will be described in the next two sections.

[13] For example, the Berkeley BIND name server is explicitly recursive and allocates very large areas of automatic storage in each recursive call.

```
                 ________________________
 (R12) -->     |                          |
               .  Parameters to my call   .
               .                          .
               |__________________________|
 (R12)+p-->    |                          |
               |    Register Save         |
               |         Area             |
               |   (16 fullwords for      |
               |      R0, ... R15)        |
 (R12)+p+      |__________________________|
   64  -->     |                          |
               |                          |
               |      Automatic           |
               |                          |
               .      Variables           .
               .                          .
               |__________________________|
               |        A(PTA)            |
               |__________________________|
 (R13) -->     |                          |
               .   (next stack frame)     .
               .                          .
               .                          .

 Here:  p = number bytes of parameters
```

Figure 7: C/370 stack-frame

4.2.1 PTA Address Preservation

When a "C" program executes within the ACP, it must be able to call A-Services and P-Services, and these calls require that R11 contain the PTA address. It might seem easy to write interface subroutines in BAL which can be called from "C" programs to restore the exact register environment that the A- and P- services expect. However, the C/370 compiler has already allocated all 16 of the general-purpose registers, so there is a problem locating the necessary PTA address from within an interface subroutine. In order to allow porting and to simplify coding, we did not want to require that the PTA address be explicitly carried down through a nest of "C" subroutines.

Our solution was to 'hide' the PTA address in the stack, at the end of each stack frame (see Figure 7 above).

Of course, the compiler makes assumptions about the format of a stack frame, and we could not modify the compiler. Fortunately, all the manipulation of stack pointers by a C/370 object program is performed by BAL macros. By simply changing these macros we could add the PTA address to the stack and augment the standard entry sequence with instructions to copy the PTA address from the preceding stack frame into its hiding place in the current frame (see TR52 [Braden85B] for details).

In any call from a "C" program, the PTA address can therefore be found at address (R13)-4. This allows us to write interface routines in BAL to call A-Services and P-Services from "C" programs.

4.2.2 Interface Subroutines

The following two functions are used to call A- and P-Services which have standard calling sequences (parameters and results in the two registers R0 and R1, and a return code in R15):

* acall()

* pcall()

However, it was necessary to develop specific little BAL interface routines for some services, either because they had a non-standard calling sequence or because some additional bookkeeping was necessary to maintain a clean interface in "C". For example, the following tiny BAL routines are available:

* abort() -- Call PABEND

* afetch() -- Return value from ATRV

* delay() -- Call PWAIT

* ihostcnv -- Call IHOST CNV#

* inet_addr -- Call IHOST NUMBER

* palloc() -- Call PCORE GET

* pattach() -- Call PATTACH

* pcgive() -- Call PCORE GIVE

* pexit() -- call PEXIT

* pfree() -- Call PCORE FREE

* ppost() -- Call PPOST

* ptaaddr -- Return current PTA address (from its hiding place in the stack).

For a full list of interface routines and a complete specification of their calling sequences, see TR52.

4.3 REENTRANCY

Much of the code within the ACP may be concurrently executed by multiple ptasks and must therefore be pseudo-reentrant. That is, when a particular ptask issues an A-Service or P-Service call which will allow another ptask to run, there must be a private copy of all variable data used by the suspended ptask.

Unfortunately, there are serious reentrancy problems with C/370. The C/370 compiler assembles all static and external variables into addresses which are bound at assembly or link-edit time. A happy result is that access to these variables is very efficient; however, these variables prevent reentrancy.

A natural and efficient programming style in "C" depends upon the use of external and static variables for communication among subroutines; without these variables, all communication must be performed via explicit parameters.

A number of possible solutions to this problem were considered.

* Separate Code Copies

  Under UNIX, for which "C" was originally designed, each new process has its own copy of the code and data areas. Presumably, we could use an analogous technique in the ACP, fetching a new copy of a "C" program module into (real and virtual) storage for each execution ptask executing it.

  Unfortunately, ICT does **not** presently support the fetching of multiple copies. It assumes that all load modules are reentrant, fetching a single copy for all concurrent execution instances. It would be possible to extend ICT to fetch multiple copies of modules which are not marked "REENTRANT", but the result would be to increase the working set size and perhaps exhaust storage if many copies of a large module were used.

* Explicit Reentrance

  Another choice would be to write every "C" program for use in the ACP to be explicitly reentrant. This generally requires that every procedure call include at least one additional parameter, the address of a (reentrant) structure containing (or pointing to) all private data.

This approach, which is also used for reentrant PL/I code, loses a little of the elegance of "C", but is feasible. We used this approach in writing the SMTP sender program USMTP, for example.

* Serialize Execution

Unfortunately, "C" code which is ported from another system will inevitably make use of static and external variables. To port such a program without massive modifications, it is necessary to serialize the use of the program module. This approach was taken with the SMTP server program SSMTP and with the BIND name server code NAMED.

In order to write explicitly reentrant code, we must also ensure the reentrancy of the required "C" library routines. In particular, there is a major reentrancy problem in the definition of stream I/O in the standard C/370 library: the I/O stream control variables are kept in an array of structures (struct _iobuf) which is an external variable and therefore link-edited with the program module. The standard I/O stream names (stdin, stdout, stderr) are preprocessor macros which translate into fixed offsets into this array. For use within the ACP, a more reentrant mechanism was required.

The standard I/O routines were changed so that the _iobuf structures are allocated dynamically at execution time. The standard I/O stream names are then translated via preprocessor macros into calls on (very short) BAL subroutines that locate the required structure element.

4.3.1 Reentrant Environment

A C/370 object program normally has a highest-level procedure named "main()". When the C/370 compiler generates code for this procedure it includes a standard CSECT of code containing a "startup" sequence to establish the "C" execution environment. This startup CSECT is emitted by the compiler as a series of BAL macro calls, to be expanded by the assembler. By simply changing these macros we could change the program environment for the ACP.

In particular, the startup macros included with C/370 issue GETMAIN and FREEMAIN SVC's to obtain and free the stack area; these macros were modified to use PCORE GET and PCORE FREE for use in the ACP. As a result, a stack area in the ACP will be freed explicitly by normal program termination and freed automatically in any other case.

At the beginning of the PCORE'd stack area, there is reentrant storage for several environmental tables needed by the "C" program. To allow the program to locate these tables, the address of the beginning of the stack area is kept in a known place in the ptask's PTA (word PTAUSER).

These reentrant environmental tables are as follows:

* Stream I/O Table

The "I/O Table" (IOTABLE) is a vector of pointers to _iobuf structures for open stream I/O streams. The first three slots in this vector correspond to the standard input, output, and error units (stdin, stdout, and stderr, respectively); the other slots are assigned as needed.

This vector is used (1) to locate the standard I/O units, and (2) to ensure that all I/O streams are properly closed when the program exits.

* Saved Environment

The library routines setexit() and reset() provide a mechanism to break out of an indefinite nest of subroutines and return to a known earlier execution state. These routines use a 12-word register save area (the volatile registers R0-R3 are not kept) which is in the beginning of the stack area.

* Execution Parameters

The startup sequence calls a library subroutine (_gtargs()) to parse the execution-time parameter string before main() is called. The ACP version of _gtargs() copies the parameter into the stack and then parses it into a series of null-terminated strings, one for each parameter item. _gtargs() then creates "argv[ ]", a vector of pointers to these strings, in the stack following the string.

* ALARM Timeout Time

The fullword BOSTIMO is used to save the next time-of-day at which a timeout should be signalled. See the signal() subroutine in TR52.

* Memory Pool Head

The fullword BOSMROOT contains the head of the chain of storage pools used by the malloc() and free() routines (see the next section).

4.4 STORAGE ALLOCATION

Within the standard "C" library, the routines malloc() and free() are used to obtain and free main storage blocks. The C/370 library implements these using GETMAIN and FREEMAIN. However, within the ACP we need to use PCORE, to ensure that storage will be freed if a ptask fails to free it.

Within the ACP, there are two levels of storage allocation routines available.

4.4.1 Wholesale Allocation

The routines palloc() and pfree() call the PCORE service directly to provide efficient allocation of main storage in units of 256-byte pages. The palloc() and pfree() routines have the same calls as malloc() and free(), respectively, and can be used in place of the standard routines by changing the names:

```
#define malloc    palloc
#define free      pfree
```

PCORE FREE requires as parameters both the address and the length of a block of storage. To provide this, palloc() prefixes the storage block with a "hidden" header of two fullwords.

4.4.2 Retail Allocation

The general storage allocation routines malloc() and free() suballocate arbitrary-sized segments from 4K pools obtained with PCORE. If the demand exceeds the storage pool, malloc() will call palloc() to obtain a new 4K pool block. All 4K pool blocks are chained together, with the start of the chain in word BOSMROOT in the stack area.

The routines use a traditional first-fit algorithm with roving pointer. Each segment is preceded by a hidden fullword header.[14]

[14] We are grateful to Robert Cole of University College London, who supplied the malloc() and free() routines which he developed for the MOS realtime system on LSI/11 CPU's.

4.5 I/O SYSTEM

The ACP environment required rewriting many of the I/O support routines from the C/370 library and changing the standard #include member "stdio.h".

It will be seen that the ACP version of the I/O library are much less general than those of UNIX. We chose to keep it simple, implementing just those facilities required for the particular "C" programs of interest. However, we have maintained the device-independent stream I/O capability of the standard "C" library [JohnRi78].

4.5.1 Streams

In order to perform I/O to particular external devices, a "C" ptask must open logical devices or streams. *The library subroutine fopen() is used* to open a stream. If successful, fopen() returns a quantity called a file pointer, which is the handle used in all subsequent I/O calls for this stream. A file pointer is actually the address of a 32-byte area whose format is defined by the "C" structure "_iobuf" (in BAL, the #IOBUF DSECT). The format and function of this area are similar, but not identical, to those of the standard "C" library routines.

The ACP version of the I/O system is built around two kinds of streams: buffered and unbuffered. Buffered streams support the character-by-character I/O calls getc() and putc() of the standard "C" library. These routines implicitly call the unbuffered I/O routines (readf() and writef()) as necessary. For unbuffered streams, the ptask calls readf() and writef() directly.

Note that "C" I/O in the ACP differs from the UNIX I/O interface in not being structured into a two-level hierarchy (stream I/O and "low-level" I/O). In the ACP, the file pointer is used as the universal handle to all I/O calls, including the unbuffered I/O calls; there is no "file descriptor". The unbuffered readf() and writef() routines in the ACP are otherwise equivalent to the low-level read() and write() routines of the standard UNIX interface.

A buffered stream is simplex, i.e., it may be opened for either reading or writing, not both. Full-duplex external devices such as TELNET connections or VTAM connections require a pair of streams, one for input and one for output. For these devices, a single call to fopen() will open both streams and return the file pointer corresponding to the output stream. Given the output file pointer, the library routine Ginfp() will return the input file pointer.

Within the ACP, there is a major restriction on stream I/O: it is strictly sequential. Thus, the standard "C" library routine fseek() was not implemented within the ACP.

4.5.2 I/O Library Routines

The following list shows the most important I/O routines used by "C" routines within the ACP. Those names followed by asterisks are compatible with routines by the same name in the standard "C" library, in terms of call and effect but not implementation. In some cases we have added restrictions, which are noted.

* fopen() *

Open a stream (or full-duplex pair of streams).

* fstopen()

Open a stream to a particular standard I/O unit (or, for a full-duplex buffered device, a pair of streams to stdin and stdout).

This call is necessary to force the allocation of the proper slot in IOTABLE for the standard units; fopen() would allocate any available slot.

* Ginp()

Given the output file pointer for a full-duplex external device, return the corresponding input file pointer.

* fclose() *

Close a specified stream. For a full-duplex pair of streams, fclose() must be invoked for the output stream; it may (but need not be) invoked for the input stream. Input and output streams may be closed in either order.

For some stream types (Telnet and VTAM connections in particular), fclose() has no effect on the underlying connection. For disk and sysout datasets, however, fclose() will do a PCLOSE and PDYNAL FREE.

* fflush() *

Force buffered output to be sent to a specified output stream.

When an output stream is closed, fclose() always calls fflush() implicitly. A ptask will need to issue fflush() explicitly for a buffered interactive stream (e.g., a TELNET connection or a VTAM connection), in order to force buffered data to be sent to a user.

* putc() *

Append a character to a specified output stream.

* fputs() *

Append a null-terminated string to a specified output stream.

* getc() *

Return next character in specified input stream. If necessary, an internal PWAIT will be performed until a character is available.

* fgets() *

Get null-terminated string, up to next end-of-line, from a specified input stream. If necessary, internal PWAIT's will be performed until the call completes.

* ungetc() *

Return the most recent input character to the input stream buffer.

* writef()

Write out a buffer of data to a specified output stream. This is the unbuffered write routine called indirectly from putc() to empty the stream buffer; it must be called explicitly for an unbuffered stream.

* readf()

Read in a buffer of data from a specified output stream. This is the unbuffered read routine called indirectly from getc() to fill a stream buffer; it must be called explicitly for an unbuffered stream.

The calls to these routines are described in more detail in a later section.

4.5.3 External Device Types

Each stream is opened to a particular "external device". The following types of external devices are supported by the present implementation:

* Telnet connection

TELNET I/O uses a full-duplex pair of buffered streams. To support this type, readf() issues an ATGET and writef() issues an ATPUT. If ATGET finds no input available, readf() issues a PWAIT for input with a 10 minute timeout.

* VTAM connection

A full-duplex pair of streams can be used to perform I/O to the VTAMTERM (VCALL) interface to VTAM. The present implementation is limited to virtual-3767 (line mode) access. These I/O calls will PWAIT internally for available input and for output to complete.

* Disk dataset

A disk dataset can be either read or written sequentially.

To open a disk I/O stream, issue:

```
FILE * fopen( dsname_ptr, dalo_ptr )
```

Here "dsname_ptr" is the address of a null-terminated dsname string, and "dalo_ptr" is the address of a structure containing DCB parameters and other information. This call will make the required PDYNAL and POPEN requests. fclose() will make the corresponding PCLOSE and PDYNAL FREE requests.

There are some significant implementation restrictions on the dataset attributes:

1. DSORG may only be PS.

2. For input, RECFM may be FB, FBA, VB, or VBA.

3. For output, RECFM may be VB or VBA; however, there is presently no mechanism to set the Carriage Control character (column 1) in a VBA dataset.

* SYSOUT dataset

Output can be written to a SYSOUT dataset with specified values for CLASS, DEST, and COPIES.

The present implementation restricts a SYSOUT dataset to RECFM=VBA and LRECL=137 or to RECFM=VB and LRECL=84.

To open a SYSOUT stream, issue:

```
FILE * fopen( "*PRINT", sop_ptr )
```

Here "sop_ptr" is the address of a structure containing the CLASS, DEST, and COPIES parameters.

When the stream is closed, fclose() will call PDYNAL TYPE=FREE. This will normally have DISP=(,KEEP), causing the dataset to be enqueued. However, the library routine fkill() will mark the stream with a bit which will cause fclose to use DISP=DELETE, deleting the dataset.

* WTPLOG file

A buffered output stream can be opened to any of the WTPLOG files -- ERROR, EVENT, ACCT, etc., using ITRACE.

* Trace buffer

A buffered output stream can be opened to a circular trace buffer. This buffer can be subsequently dumped to a WTPLOG file by calling the ITRACE service.

* TCP connection

This is an unbuffered full-duplex I/O stream.

For input, readf() issues ARECV MOVE. If no data is available, it issues an internal PWAIT for INPUT, CLOSE, and TIME. The timeout is an assembly constant currently set at 10 minutes.

For output, writef() issues an ASEND and then PWAIT's for OUTPUT completion. In the current implementation, the TCP PUSH bit is always set. It therefore provides poor TCP throughput unless a very large buffer is passed to writef().

For this type, fopen() will call the ALSTN and AOPEN services to actively open the TCP connection, while fclose() will issue ACLOSE WAIT.

* UDP association

This is an unbuffered full-duplex I/O stream, using the datagram-oriented I/O routines udpsend() and udprecv() rather than the standard readf() and writef().

4.5.4 Newlines and Tabs

In normal "C" usage, a newline is denoted by a LF character in a text string. C/370 preserves this usage, except that the LF, like all character constants, is represented in the EBCDIC code, not ASCII.

To allow easy porting of existing code, it was desirable to preserve the convention of LF as an end-of-line character within "C" code. However, within the rest of the ACP a newline is represented by the EBCDIC NL character. Furthermore, in record-oriented external devices like disk files, there is no explicit newline character. This difference in end-of-line conventions was handled in different ways for input and for output.

* Input

Suppose the getc() library routine is called requesting the next character character from an input stream. As in the standard "C" library, our getc() is actually the macro [KernRi78]:

```
  #define getc(p)

          (--(p)->_cnt >= 0 ? *(p)->_ptr++:_fillbuf(p))
```

This macro tries to get the next character from the stream buffer, decrementing the counter _cnt and incrementing the buffer pointer _ptr. However, if the buffer is empty (_cnt < 0), it calls _fillbuf() to refill the buffer.

The primary function of _fillbuf() is to call the unbuffered read routine readf() to read the next chunk of data into the buffer. In this chunk of data, an end-of-line will be represented by the character NL. For a disk dataset, readf() will actually supply the next logical record and add an explicit NL.

After calling readf(), _fiilbuf() calls a BAL routine (___tr()) to translate any NL's in the buffer into LF's.

* Output

The putc() macro has been replaced by a BAL routine which performs essentially the same function: if there is room, add the character to the buffer, increment _ptr, and decrement _cnt. If there is not room, OR IF the character is a LF, call the _flshbuf() routine to empty the buffer.

_flshbuf() tests the type of the stream. For a record-oriented stream, it always calls writef() to write the next record, and DELETES a LF. For other streams (e.g., TELNET), it calls writef() only if the buffer is full; otherwise, it deposits the LF in the buffer buffer and returns.

4.6 PORTING DIFFICULTIES

This section will summarize the difficulties we have encountered in porting UNIX programs into the ACP.

* Large Source Files

Unfortunately, the C/370 compiler seems to have some internal overflow problems with very large source files, getting OC4 ABENDs in the code generation step. We have not pinned down the exact nature of the overflow, but segmenting the source files to less than about 2000 lines of "C" code seems to avoid these ABENDs.

* Symbol Ambiguities

C/370 has a fault inherited from its ancestors: it allows arbitrary-length symbols but only distinguishes them by the first 8 characters. Where a source program contains two symbols which differ only after 8 characters (e.g., "gethostbyname" and "gethostbyaddr"), one or both must be changed consistently throughout the source program.

Unfortunately, the 8-character limit is true of the C/370 preprocessor step as well as the compilation steps, so macros cannot be used to make consistent name changes where necessary.

* Operating System Differences

There are serious differences between UNIX and ICT as operating systems. These differences ideally show up only in the highest-level procedure of the program, which often requires significant changes for the ACP environment.

Here are some examples of system-related problems.

1. The UNIX fork() primitive cannot reasonably be translated into the ACP. The ICT PATTACH service does create an independent thread, but it cannot have its own address space. It is not sufficient to make a copy of the caller's stack, since the stack contains absolute addresses pointing to the original copy.

2. UNIX programs often include file names which do not conform to OS/MVS dataset naming conventions.

3. UNIX programs invoke each other using Shell commands, which have no equivalent within the ACP.

4. There are unavoidable differences between the UNIX and MVS file systems. For example, the UNIX unlink() function has no exact equivalent. No call to **fseek()** can be used within the ACP; it is not possible even to **"rewind"** an input or output file.

* Library Differences

The current "C" library differs in a few ways from the portable "C" library included with C/370.

1. Printf() features.

   New features appear in printf() in various versions of UNIX. Examples we have seen are "%+11d", "%m", "%r", "%#x", and "%*d". We have added these features to the printf() routines used in the ACP.

2. index()

Berkeley has changed the name of the strchr() function to index(), conflicting with the C/370 index() routine.

3. atoi()

This function does not exist in the portable C library, although it can be defined trivially using the strtol() subroutine.

* Static procedures

There seems to be a C/370 compiler bug which causes garbage labels to be emitted for static procedures. Removing the 'static' attributes (from the procedures -- static works fine for data) avoids this compiler bug.

* External Symbol Conflicts

The C/370 library routines are compiled/assembled in groups, so each load module typically has a number of entry points. This sometimes leads to accidental conflicts between an external symbol in the source program and some C/370 library routine that happened to be included with a routine that was needed.

* Tab Characters

UNIX programs often use the HT (Horizontal Tab) character in their formatted output, assuming that the UNIX system will expand them in a standard manner when data is sent to a printer or display device.

Within the ACP, the correct way to handle HT's would be to map them into the appropriate number of blanks within writef(). This would be done for certain external device types -- disk datasets, VTAM connections, and log files. However, this would add considerable complexity to writef(), and to date we have not implemented the necessary mechanism; we have chosen to modify the source programs instead. We regard HT handling as an unresolved compatibility problem.

Chapter 5

RELEASE 1.6 FEATURES

This chapter summarizes the major features of Release 1.60.

* LNI Revision

This has been discussed in a previous chapter.

* Host Lookup Changes

Changes to prepare for a name resolver were discussed in a previous chapter.

* USMTP Update

Release 1.6 is the first to include the SMTP modules, most of which are written in "C". During the present contract, the USMTP module (User SMTP) was rewritten in order to:

1. Decrease the size of the load modules by obtaining most working storage dynamically.

2. Include a complete parser for the RFC822 header syntax, to enable the SMTP code to properly map arbitrary addresses. This mapping is desirable because user mail systems for IBM equipment tend to use cryptic and non-mnemonic account names, rather than person names, for mailboxes. The SMTP implementation is prepared to map these little horrors to/from real person names for Internet consumption.

* Multihoming

The redesign of the LNI interfaces to allow multiple LNI's raised the possibility of installing the ACP on a multihomed host, i.e., a host with multiple IP addresses. To support such local multihoming, it was necessary to generalize the IP Program (IPP) to effectively make the choice of local address a parameter of each connection/association.

This generalization required the revision of a few internal interfaces and control blocks. There was already a root control block (P3CB) which contained the local host address. With a few changes, it was possible to allow multiple P3CB's, one for each multihomed address.

* Gateway Module

Since Rel 1.6 supports multihoming, some installations may want to use the ACP as a dumb gateway ("dumb" because it does not support the Exterior Gateway Protocol, EGP). To handle the variety of such problems, Release 1.6 includes a module named GATEWAY.

When the ACP receives a packet which is not destined for the local host (either the packet's destination address does not match any local address, or the packet contains an active source route), the entire packet will be passed (without reassembly) to GATEWAY. GATEWAY is structured as a higher-level protocol module, as appropriate for a gateway function.

The GATEWAY module that is distributed with Release 1.60 will forward a packet if its destination network matches one of the networks to which the ACP is locally attached; otherwise the packet will be discarded and a message ("NO ROUT 4PKT") logged.

Obviously, more general algorithms are possible, e.g., implementation of some locally-defined "Interior Gateway Protocol" (IGP). This must be left for future development.

* Gateway Pinging

When the ACP is used to drive a local Ethernet, it will be necessary to "ping" local gateways to ensure that they are up. The basic mechanism for this pinging will be a new permanent ptask, GWYPING. This ptask has not been written yet; however, Release 1.6 contains the "hooks" which will be needed to interface to the rest of the ACP.

We envision GWPING operating in the following manner.

1. GWYPING will be driven primarily by the dynamic table of active Internet hosts and gateways, IHTAB. GWYPING will ping all the gateways on this table at a slow rate (e.g., once every 20 seconds). Note that this table lists only those gateways to which packets are actually being sent by the ACP. This background pinging will detect gateway outage during TCP idle periods, and it will establish average round-trip times for each gateway.

2. If GWYPING decides that a gateway is down (it has received no reply after a set number of successive pings), it will pass the IP address of the gateway to the new service INTERNET GWYDOWN.

   GWYDOWN will cause the IPP to behave exactly as if a HOST DEAD message had been received for that host from the local network. In general, this will not kill a TCP connection; it will only cause the normal mechanism to choose another primary gateway from the list in the ACP.

3. When TCP does the second or later retransmission, it calls a new service INTERNET HOSTPROB, to notify the IPP that there may be a problem with communication to the specific remote host.

   HOSTPROB turns on a "problem" bit in the IHT entry for the host in question, and tries to issue a PPOST to awaken the GWYPING ptask (at present, there is no such ptask, so the PPOST is skipped).

   When it is awakened and finds an IHT entry with the "problem" bit on, GWYPING will ping that gateway at a relatively high frequency (e.g., once every 2 seconds). The round-trip average accumulated by the background pinging will provide a realistic timeout criterion. Again, if it decides that the gateway is down, GWYPING will call INTERNET GWYDOWN to force a re-routing.

   If GWYPING decides the gateway is UP (e.g., it receives more than 2 out of 5 pings back), it will simply turn off the "problem" bit in the IHT and revert to background pinging of that gateway.

* Subnet Support

The ACP now contains the (minimal) support for subnets as defined in RFC 950 [JMogul85]. There is a new 32-bit mask field in the configuration table (ACPCONFG) to define the bits corresponding to the subnet field. This value will be OR'd with the normal network mask to obtain the effective network mask (called "my_ip_mask" in RFC 950). If there are no subnets, the subnet mask will be zero.

* MVS/XA Compatibility

OAC recently installed the IBM operating system MVS/XA, which is the 32-bit addressing version of OS/MVS. The ACP is now executing (in 24-bit addressing mode) under MVS/XA, but will still execute under the earlier MVS/SP operating system. This change exposed a few system dependencies within ICT.

* Broadcast Facility

The ACP now has a broadcast facility, which allows an operator to send a message to all User TELNET and Server TELNET sessions, or to a particular session. The operator interface to the broadcast service is a a "C" program named BCAST.

* Expanded Well-known ports

The range of "well-known" ports was expanded from 1:255 to 1:4095, and the private experimental ports within the ACP were set to 2021 (FTP), 2023 (TELNET), 2025 (USMTP), and 1023 (VM Full screen). A new SETA symbol in INPCONFG defines the upper limit of the WKP range.

* Indefinite Listen Changes

To support continuous servers (e.g., USMTP and NAMSERV), several changes were been made to indefinite listens, i.e., passive connection requests which contain "wild" remote host and/or port numbers. For example, the IPP was changed so a definite ALSTN call will not match a pending indefinite listen.

* Host Unreachable Softened

The treatment of the ICMP Host Unreachable message by TCP was changed. Previously, TCP within the ACP treated Net Unreachable as a "soft" error, but Host Unreachable as a permanent error. However, we had the experience of trying to do a long FTP to a host whose network interface was bouncing up and down, resulting in frequent but temporary Host Unreachable messages.

Host Unreachable will now prevent a new TCP connection from opening but will not kill an existing connection (although retransmission timeout may kill it). There will always be a log entry for the Host Unreachable, so we can be aware of flaky interfaces.

* Saved-trace Mechanism

Over a number of years, various debugging facilities have been put into the ACP, and those that were useful have been further developed while those which were not useful have atrophied.

For example, the original connection trace buffer mechanism failed to provide useful debugging information and was allowed to atrophy. During the present contract, we developed a new approach to connection tracing, the "saved trace". During normal operation, the data stream is saved in binary in a circular buffer. Only if a severe error occurs will the high-overhead operation be performed of formatting the contents of the buffer to an ACP log.[15] The SMTP user and server modules implement saved traces of their SMTP protocol streams, while TCP implements a saved trace of its connection data. These traces have been useful in finding both local and remote protocol problems.

Exercising the saved-trace mechanism for TCP revealed both bugs and design problems with trace buffers associated with connections/ associations. The problems had to do with race conditions at close time.[16] These problems should be cured in Release 1.60.

[15] This approach was actually inspired by a debugging technique in the SMTP server developed by the UCLA Computer Science Department for use under UNIX: write a trace history into a file, and discard the file if no error occurs. UNIX files have low overhead, but under OS/MVS datasets do not. However, within the ACP the use of a circular buffer provides an analogous facility with acceptable overhead.

* Connection Statistics

Code was added to gather and report a fairly complete set of statistics on TCP connections. The data is written into the ACP event log whenever a TCP connection is closed.

[16] If the TRB appears ahead of the corresponding TCPB in the all-CCB chain, the TRB will be closed first; but the process of closing the TCPB may well result in trace entries. Conversely, trace buffers were not always being freed.

The data consists of two lines, one for input and one for output. Here are two typical lines (shown folded to fit on this page):

```
IN <  378=#SEG    0=#DSG 378=ACKS   0=#DSC       4=#DBY
                    ...  0=MAX    0=LIM    0=AVG    0=QMAX

OUT>  197=#SEG 194=#DSG   3=ACKS  12=#REX 182720=#DBY
                    ...964=MAX 964=LIM 941=AVG 4.33=RTT
```

The quantities listed here are defined as follows:

1. #SEG

   The total number of segments sent/received.

2. #DSG

   The number of segments that carried data octets (segments with only SYN, FIN, or ACK bits are not counted).

   In the input side, the packets which were discarded because their sequence numbers fell outside the window are also not counted. In the output side, each retransmission of a data segment is counted in #DSG.

3. ACKS

   The number of segments which did not carry any data octets; it includes empty ACK packets, SYN packets, and FIN packets. These numbers are derived:

   ACKS= #SEG - #DSG.

4. #DSC

   The number of input segments which are discarded because their sequence numbers were "unacceptable", i.e., fell outside the current receive window. Normally, these will represent unnecessary retransmissions by the remote host.

5. #REX

   The total number of extra (data) segments which were sent in retransmissions.

6. #DBY

The total number of data bytes sent or received; it does not count the octet space consumed by the SYN and FIN bits.

7. MAX

The largest number of data octets sent or received in a TCP segment on this connection. It does not count the octet space consumed by the SYN and FIN bits.

8. LIM

The maximum number of data octets per segment which could have been sent or received on this connection.

In the receive side, it is the smaller of the reassembly buffer size (less the TCP header size) and the size of the receive circular buffer. On the send side, it is the maximum segment size (576, unless the remote host changes it with the Maximum Segment Size option), less the TCP header size.

9. AVG

The average number of data octets per data segment sent or received.

On the receive side, this is computed as #DBY/(#DSG+#DSC). On the send side, it is computed as #DBY/#DSG. Thus, retransmissions and discarded packets tend to reduce the average.

10. QMAX

The largest depth of queuing of out-of-order segments in the input side.

11. RTT

The measured round-trip delay (seconds) at the time the connection was closed.

From these figures, a great deal can be inferred about the efficiency of different TCP implementations, and about the effect of long path delays and packet losses in the Internet.

If IP reassembly was required on the connection, there will be an additional log line reporting the number of packets received and the number of complete segments reassembled.

Chapter 6

CONCLUSIONS

Release 1.6 of the UCLA ACP represents the end of the DARPA-funded development of OS/MVS host software for the Internet protocols. The ACP is likely to see considerable operational usage in DDN, so continuing maintenance and extension will be necessary.

The Internet is a continually-changing enviroment, and we already know of a number of areas for short-term development in the ACP.

* Finalize the name resolver/name server code

  Experimental modules exist, but more time and work will be needed to make the name server/name resolver system a fully reliable and efficient replacement for the static host tables.

* Implement MX in USMTP

  The USMTP module needs to employ the name server system to look up mailbox hosts, based upon MX records.

* Implement UNAME and MNAME Options in GETHBY

  These options would make the ACP user interface easier to use, since the user would be able to enter a partial host name and have it completed.

* Implement 1822L Access Protocol

  DDN hosts are going to need the 1822L access protocol to the DDN, to support logical addressing and multiplexed interfaces. This represents a modest extension to the IMP1822 LNI module.

* Remote Job Entry

  Some Internet facility for remote job entry will be needed very shortly, to support the new scientific user community that will be added by NSF and DARPA.

In addition, there are a number of longer-term development areas, whose importance is difficult to assess at present.

* Gateway Pinger

Earlier in this report we outlined the design for a gateway pinger ptask (GWYPING), which will be necessary if the ACP is connected to an Ethernet on which there are gateways to further nets.

* Gateway and EGP

As discussed earlier, the GATEWAY module of the ACP could be expanded to have real routing functions. For this to be useful, however, EGP (Exterior Gateway Protocol) will be needed within the ACP. A good possibility would be to port an existing "C" EGP.

* I/O Performance Improvements

There are several possible ways to improve the I/O performance of the ACP. Investigation and experimentation are necessary to decide upon the best course.

1. Use VTAM for I/O to channel interface.

   A channel interface would be used that emulated an IBM control unit, such as a 3274. VTAM's I/O may be more efficient than the ACP's, since VTAM is "authorized".

2. Fast-Path I/O

   If an installation is willing to let the ACP run "authorized", it may be possible for the ACP to use a more efficient form of I/O, called "fast-path".

3. Packet Aggregation

   If the local network interface aggregated input packets and disaggregated output packets, the number of I/O interrupts could be substantially reduced, making a corresponding dramatic improvement in CPU utilization in the ACP.

* Internet Congestion Control

The Internet research community cannot defer much longer a solution for congestion. Any solution will have some impact upon the host software, requiring ACP changes.

A minimal approach would be to breathe life into the ICMP Source Quench message. A more significant change would be to use "metered" output at the IP level. The challenge within the ACP will be to meter the output without creating unacceptable timer-event overhead.

* User FTP Rewrite

The present User FTP processor, which runs under TSO, has a user interface which is very general but not very convenient.

An attractive alternative would be to write a new User FTP, with a user-server model interface, in "C" to execute within the ACP.

* Direct User TELNET Enhancement

The ACP routine (VTAMAPPL) that handles direct terminal access to User TELNET should be enhanced to support multiple sessions, session logging, reading an input file, etc. Then the TSO TELNET program, which is a maintenance problem, can be dropped.

* ISO Protocols

In the (not so distant) future, there will be a desire to switch from TCP/IP to the corresponding ISO protocols. This should be much less disruptive a change than from NCP to TCP/IP. It will generally only require replacing the IPP and TCP routines, leaving most of the ACP intact.

A good possibility will be to port a "C" version into the ACP. One serious problem will be to accomodate the variable-length addresses of ISO IP. When we first installed TCP and IP into the ACP, we had in mind that IP addresses might eventually become variable-length. Although 32-bit host numbers have crept back into many internal ACP interfaces during the succeeding nine years, a change to variable-length addresses is far from hopeless.

REFERENCES

BBN1822 BBN, "Specifications for the Interconnection of a Host and an IMP". Report 1822, Bolt Beranek and Newman, Cambridge, Massachusetts, revised May 1978.

BBN1822L Malis, Andrew G., "The ARPANET 1822L Host Access Protocol". RFC878, December 1983.

Braden85A Braden, R., "Continued Development of Internet Protocols under the IBM OS/MVS Operating System -- Final Technical Report". Technical Report TR46, Office of Academic Computing, UCLA, January 1985.

Braden85B Braden, R., "'C' Code Within the ACP". Technical Report TR52, Office of Academic Computing, UCLA, October, 1985.

Braden85C Braden, R., "SMTP Implementation Within the UCLA ACP". Technical Report TR49, Office of Academic Computing, UCLA, December 1985.

Braden86A Braden, R., "The Local Network Interface in the OS/MVS ARPANET Control Program". Technical Report TR51, Office of Academic Computing, UCLA, January 1986.

Braden86B Braden, R., "Datagram Protocol Implementations Within the UCLA ACP". Technical Report TR55, Office of Academic Computing, UCLA, January 1986.

Braden86C Braden, R., "UCLA ARPANET Control Program for OS/MVS Using Internet Protocols TCP/IP -- Technical Overview". Technical Report TR35, Office of Academic Computing, UCLA, (revised) February 1986.

Braden86D Braden, R., "Introduction to the UCLA ACP -- Internet Software for IBM OS/MVS". Technical Report TR48, Office of Academic Computing, UCLA, February 1986.

ISOIP ISO, "Protocol for Providing the Connectionless-Mode Network Services". ISO DIS 8473, reprinted as RFC 926, December 1984.

JMogul85 Mogul, J. and Postel, J., "Internet Standard Subnetting Procedure". RFC950, August 1985.

JohnRi78 Johnson, S. and Ritchie, D., "Portibility of C Programs and the UNIX System". Bell System Technical Journal, Vol. 57, #6 Part 2, July-August 1978.

KernRi78 Kernigan, R. and Ritchie, D., "The C Programming Language". Prentice-Hall, Englewood Cliffs, NJ, 1978.

Mockap83 Mockapetris, P., "Domain Names -- Implementation and Specification". RFC883, November 1983.

Postel81A Postel, J., "The DoD Standard Transmission Control Protocol". RFC793, September 1981.

Postel81B Postel, J., "The DoD Standard Internet Protocol". RFC791, September 1981.

Postel81C Postel, J., "Internet Control Message Protocol". RFC792, September 1981.

Postel81D Postel, J., "File Transfer Protocol". RFC765, September 1981.

Postel81E Postel, J., "Simple Mail Transfer Protocol". RFC821, August 1981.

Rivas84 Rivas, L., and Ludlam, H., "ACF/VTAM Usage by the ARPANET Control Program". Technical Report TR37, Office of Academic Computing, UCLA, (revised) October 1984.

Stein84 Stein, M., Rivas, L., and Ludlam, H., "ICT Version 2 Pseudotask Services and Macro Instructions". Technical Report TR38, Office of Academic Computing, UCLA, (revised) December 16, 1984.

TR21A Braden, R., Rivas, L., and Ludlam, N., "Programming User-Level Protocol Processes for ARPANET ACP (REVISED)". Technical Report TR21A, Office of Academic Computing, UCLA, Revised November 1984.

TR42 Rivas, L., Braden, R., and DeLaRoca, D., "Installation and Operation Guide for the UCLA ARPANET Control Program -- Release 1.60". Technical Report TR42, Office of Academic Computing, UCLA, Revised December 1984.